Language Generation and Verification in the NRL Protocol Analyzer

Catherine Meadows
Center for High Assurance Computing Systems
Naval Research Laboratory
Washington, DC 20375
USA

Abstract

The NRL Protocol Analyzer is a tool for proving security properties of cryptographic protocols, and for finding flaws if they exist. It is used by having the user first prove a number of lemmas stating that infinite classes of states are unreachable, and then performing an exhaustive search on the remaining state space. One main source of difficulty in using the tool is in generating the lemmas that are to be proved. In this paper we show how we have made the task easier by automating the generation of lemmas involving the use of formal languages.

1 Introduction

The NRL Protocol Analyzer is a tool for proving security properties of cryptographic protocols, and for finding flaws if they exist. In its most basic form, it is a search tool. A goal (usually an insecure state) is presented to it, and it attempts to find all paths to that state. However, exhaustive search in itself is not an adequate means of verifying the security of cryptographic protocols. This is because the state space is assumed to be infinite. For example, it is necessary to assume that an unbounded number of executions of a protocol may have taken place, and that a principal can be engaging in an arbitrarily large number of protocol executions at any given time. Moreover, for the purposes of analysis, very large sets, such as the number of keys available, or the number of words that can be produced by encrypting a word over and over again, are assumed to be infinite.

In order to deal with these problems, we have developed several ways in which users of the Analyzer can prove lemmas about the unreachability of infinite classes of states. One of the most important of these involves induction on formal languages. The user defines a formal language and uses the Analyzer to prove that, if an intruder trying to break the protocol has found a word in that language, then the intruder must have already known a word in that language. This, together with the fact that the intruder knows no words in the language initially (if that is the case), can be used to prove inductively that the intruder can never learn a word in that language. The procedure for proving a language unreachable has been automated, and is documented in [5].

Although automation of the language verification procedure was helpful, until recently it was up to the user to define the language his or her self. This was not an easy procedure for complicated protocols, and required close inspection of Analyzer output, as well as of the output of the language verifier whenever it failed in a proof. Moreover, it was often possible to define a language that could be proved unreachable, but was actually somewhat smaller than necessary. It was difficult to detect when this had occurred, but failure to prove the largest possible language unreachable could result in an unmanageably large search space.

Fortunately, it is possible to describe an heuristic procedure for defining formal languages that avoids many of these problems, and this has been automated in the most recent version of the Analyzer. Although this procedure does not guarantee the largest possible language, we have used it to prove unreachability of languages that are large enough to be useful, and we have found that it saves a significant amount of labor. In this paper we describe how this procedure works.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1996		**00-00-1996 to 00-00-1996**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Language Generation and Verification in the NRL Protocol Analyzer	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**14**	

2 How Languages Are Used in the Analyzer

The NRL Protocol Analyzer is written in Prolog and relies upon equational unification. It employs the worst-case model used by Dolev and Yao [1] in which the network is controlled by a hostile intruder who can read all message, destroy messages, and create or modify messages. Since the intruder can read all messages, any message sent may be assumed to have been received by the intruder, and since the intruder controls what messages are received, any message received is assumed to have been sent by the intruder. Thus we can think of the protocol as an algebraic system that is manipulated by the intruder. We also assume that the intruder may be a legitimate participant in the protocol, and so has access to some (but not all) of the secret keys used, and also has the ability to perform operations such as encryption available to legitimate participants. As do Dolev and Yao, words in the Protocol Analyzer model obey a set of *rewrite rules* specified by the user. For example, the user may want to specify that encryption of a word with a key, followed by decryption with the same key, reduces to the original word.

In the Protocol Analyzer, words sent in messages or stored in local state variables are represented by terms that are made up of function symbols, constants and variables. A protocol itself is specified as a set of state transitions in which the input state is described by messages received and the values of local state variables, and the output state is described in terms of messages sent and new values of local state variables. Actions local to the intruder, such as the intruder's performing encryption or decryption, are represented internally in the same way. States are specified in terms of a set of words known by the intruder and sets of local state variables and their values. An example of a local state variable would be one holding the key that an honest principal is using to converse with another. An example of an insecure state would be one in which that state variable contains the word K and K is known by the intruder.

The Protocol Analyzer finds a complete description of all states preceding a specified state in the following way. For each state transition a subset O of the output is paired up with a subset S of the state description. The Analyzer uses a narrowing algorithm [7] to find a complete set of substitutions Σ to the variables in O and S such that the words in O can be made equal to the words in S by the application of rewrite rules. By complete, we mean that, if τ is a substitution such that τO is reducible to τS, then there is a σ in Σ such that $\tau = \mu\sigma$ for some substitution μ. The preceding state consists of the input to the state transition and the part of the state description that was not used to produce O.

Languages can arise when we attempt to find out how the intruder can find a word, and we find ourselves in an infinite regression. Consider the following very simple protocol, with two rules [1]:

Encrypt-Decrypt Protocol

Protocol Rule 1
If the intruder knows X and Y, then he or she can find e(X,Y), where e(X,Y) denotes the encryption of Y with key X.
Protocol Rule 2
If the intruder knows X and Y, then he or she can find d(X,Y), where d(X,Y) denotes the decryption of Y with key X.

The words used in the encrypt-decrypt protocol also obey two rewrite rules d(X,e(X,Y)) $\rightarrow Y$, and e(X,d(X,Y)) $\rightarrow Y$.

Suppose that we want to find all the words that the intruder can know. We ask how the intruder can find Z, where Z is a variable that can stand for any irreducible word. Using Protocol Rule 1, the Analyzer will tell us that this can be done if the intruder knows d(W,Z) and W for some W, or if Z = e(X,Y) and the intruder knows X and Y. We label these solutions 1 and 2. Using Protocol Rule 2, the Analyzer will tell us this can be done if the intruder knows e(W,Z) and W for some X, or if Z = d(X,Y) and the intruder knows X and Y. We label these solutions 3 and 4.

Suppose that we continue our search on solution 3 and ask how the intruder can find e(W,Z). Using Protocol Rule 1, the Analyzer will tell us that it can be found if the intruder can find $X = W$ and $Y = Z$ (solution 3.1), or if the intruder can find $X = X_1$ and Y = d(X_1,e(W,Z)) (solution 3.2), in which case e(X,Y) = e(X_1,d(X_1,e(W,Z))) will reduce to e(W,Z). Next, using Protocol Rule 2, the Analyzer will tell us that e(W,Z) can be found if the intruders can find $X = X_1$ and Y = e(X_1,e(W,Z)) (solution 3.3), in which case d(X,Y) = d(X_1,e(X_1,e(W,Z)).

We can rule out the solution 3.1 found using Protocol Rule 1, since it requires the intruder to know Z, which is the word he or she is trying to find. Thus, we only need to know how the intruder can find the words

[1] It is actually trivial to verify that the intruder can't learn *any* words in this protocol, since the intruder knows no words initially and every rule that produces a word requires that the intruder knew a word previously. Thus the language technique is overkill here. However, we find the simplicity of this protocol makes it helpful as an initial example

in the second two solutions. For 3.2 and 3.3, if we ask the Protocol Analyzer how to find d(X_1,e(W,Z)) and e(X_1,e(W,Z)), the reader can verify that the intruder can find the first if he or she can find e(X_2, d(X_1,e(W,Z))) or d(X_2, d(X_1,e(W,Z))), and the second if he or she can find e(X_2, e(X_1,e(W,Z))) or d(X_2, e(X_1,e(W,Z))).

If we keep applying the Protocol Analyzer, we will generate ever longer and longer words in this fashion. Thus, our search will be made easier if we can prove that, whenever Z is a word not already known by the intruder, then it is impossible for the intruder to learn e(X,Z) for any X.

We note that the patterns of words we obtained by looking for e(X,Z) showed a certain regularity. This leads us to define the following language **A**, whose definition is dependent upon the state of the intruder's knowledge:

1. **A** $\rightarrow$ e(**L**,**K**), where **L** is the set of all irreducible words and **K** is the set of all irreducible words not currently known by the intruder.
2. **A** $\rightarrow$ e(**L**,**A**)
3. **A** $\rightarrow$ d(**L**,**A**)

We now prove that **A** is unreachable by taking each language rule of **A**, substituting variables for the terms, running the Protocol Analyzer on the resulting words, and examining the words that must by input by the intruder. In each case, we try to determine that one of these words must also belong to the language. We will illustrate this procedure by showing how we show that the intruder's learning a word satisfying the second language rule implies that the intruder most already know a word belonging to **A**. The procedure for the other two language rules is similar.

We take the word e(C,B), where B is assumed to be a member of **A**. Applying Protocol Rule 1, which says that if the intruder can produce X and Y, then he or she can produce e(X,Y), gives us two solutions. In the first solution, C is unified with X and B is unified with Y. In the second, Y is unified with d(X,e(C,B)). The output e(X,d(X,e(C,B))) reduces to e(C,B). The first solution requires the intruder's previous knowledge of B, which is assumed to be a member of **A**. We now consider the second solution. This requires the intruder to know Y = d(X,e(C,B)). But, since e(C,B) is a member of the language **A** according to the second language rule, d(X,e(C,B)) belongs to **A** by the third language rule. Thus this solution requires the intruder's previous knowledge of a member of **A**.

Applying the Protocol Rule 2, which says that if the intruder knows X and Y, then he or she can learn d(X,Y) gives us one solution, in which Y is unified with e(X,e(C,B)) giving output d(X,e(X,e(C,B))) reducing to e(C,B). Thus the intruder most know X and Y = e(X,e(C,B)). Since e(C,B) is a member of **A**, e(X,e(C,B)) must belong to **A** by the second language rule, and we are done.

This procedure of proving a language unreachable has been automated in the Protocol Analyzer.

3 Notation and Definitions

In this section we outline some of the basic notation and definitions used in the rest of the paper.

3.1 Elementary Definitions

Definitions: Let X be a term. The *size* of X is the number of subterms of X.

Thus, the size of g(Y) is 2, since g(Y) and Y are the subterms. Likewise, the size of g(Z,b(t),r(s(q))) is 7.

Definition: Let X be a term and Y be a subterm of X. We define an *occurrence* ω of Y in X as follows. If $Y = X$, then the occurrence of Y in X is ϵ. If Y is the i'th argument of X, then the occurrence of Y in X is i. If the occurrence of Z in X is $i_1...i_k$, and Y is the j'th argument of Z, then the occurrence of Y in X is $i_1...i_k.j$. If $\omega = i_1...i_k$ is an occurrence, we call each i_j a *component* of ω.

Note that there can be more than one occurrence of a subterm in a term. Thus, if X = f(g,h(b),t(g)), g occurs at 1 and 3.1.

We can also compare occurrences, as follows.

Definition: Let ω_1 and ω_2 be two occurrences. We say that $\omega_1 > \omega_2$ if, either the number of components of ω_1 is greater than that of ω_2, or ω_1 and ω_2 have the same number of components and $i < j$, where i and j are the first nonequal components of ω_1 and ω_2, respectively.

Thus, for example, $1.1.1 > 3.2$, and $3.2 > 3.1$.

Definition: A *substitution* is a function from variables to terms. If a substitution assigns term T to variable V, we represent this by V/T. The identity substitution is designated by ι.

Definition: Let X and Y be two terms. We say that a substitution σ is a *unifier* of X and Y $\sigma X = \sigma Y$. We say that σ is a *most general unifier*, or mgu, of X and Y if σ is a unifier and, for any other unifier τ, $\tau = \mu\sigma$ for some μ. Most general unifiers are unique up to renaming of variables.

The following definition is somewhat nonstandard, but we use it because it describes the type of unification that is used in the Protocol Analyzer.

Definition: Let X and Y be two terms, and let **E** be a set of equations. We say that σ is a *left-handed unifier of X and Y with respect to* **E** (or simply a left-handed equational unifier of X and Y when we can avoid confusion), if σX can be made equal to σY by applying the equations from **E** to σX.

This differs from the usual definition of equational unifier, in which the equations can be applied to both terms being unified. However, we are interested in the case in which **E** is a set of reduction rules, and σY is assumed to be irreducible.

3.2 Definitions Related to the Analyzer

A protocol is specified in the Analyzer as a set of state transition rules. These are stored as Prolog clauses. We also assume, as is the case for Prolog, that quantification of variables in rules is existential. Thus, whenever a rule is used, its variables are renamed, so that substitutions made to variables in one use of the rule have no relation to substitutions made in any subsequent use of the rule.

The Analyzer is used by having the user specify a goal G consisting of words to be learned by the intruder, values of local state variables, and/or a sequence of events that should have occurred. The Analyzer returns a set of *solutions for G*. Each solution consists of an *output state S* (derived from the output of a state transition rule), a left-handed equational unifier σ_S of a subset T of S and a subset H of G, and an *input state S'* that immediately precedes $\sigma_S G$ consisting of the $\sigma_S R$ where R was the input of the rule, and any elements of $\sigma_S G$ not in $\sigma_S H$. Note that, as we saw from the example in Section 2, the output of a rule can have more than one left-handed equational unifier with a goal; thus more than one solution can be generated from a single rule. The Analyzer can be used to query S' or a portion of it; it will return a set of solutions as before, each consisting of a state S'', a left-handed equational unifier $\sigma_{S''}$ of S'' and S', and an input state S'''.

Solutions are referred to as follows. Suppose that a goal G_N is identified by an integer N. The solutions for G_N are identified by $N.1$, ..., $N.k$. The solutions found for $N.i$ are identified by $N.i.1$, ..., $N.i.n$, and so on. If $N.i_1.....i_t$ identifies a solution, we refer to the state output by the rule that produced $N.i_1.....i_t$ as $S_{N.i_1.....i_t}$, the input state to the solution $T_{N.i_1.....i_t}$ and the the restriction of the left-handed equational unifier of $S_{N.i_1.....i_t}$ and $T_{N.i_1.....i_{t-1}}$ to the variables in $T_{N.i_1.....i_{t-1}}$ as $\sigma_{N.i_1.....i_t}$. We refer to $N.1$, ..., $N.k$ as the *index* of the solution $S_{N.i_1.....i_k}$.

To see how this works, in our example in Section 2, our original goal was the state in which the intruder knew the word Z. Suppose that we assigned that goal the integer 1. The solutions we generated by trying to find the word Z would be indexed as 1.1,1.2,1.3, and 1.4. When we asked how to find the word e(W,Z) in solution 1.3, the answers we got would be indexed as 1.3.1, 1.3.2, and 1.3.3.

Let $T_{N.i_1.....i_t}$, $T_{N.i_1.....i_{t-1}}$, ..., G_N be a sequence of goals found by the Analyzer, where G_N is the original goal. Let $\tau_{(N.i_1.....i_t,N.i_1.....i_j)}$ denote the composition of $\sigma_{N.i_1.....i_j}$, $\sigma_{N.i_1.....i_{j+1}}$, through $\sigma_{N.i_1.....i_t}$. Then $T_{N.i_1.....i_t}$, $\tau_{(N.i_1.....i_t,N.i_1.....i_t)}T_{N.i_1.....i_{t-1}}$, $\cdots$, $\tau_{(N.i_1.....i_t,N.i_1)}G_N$ represents a path through the protocol. That is, it is possible to proceed from the state described by $T_{N.i_1.....i_t}$ to the state described by $\tau_{(N.i_1.....i_t,N.i_1.....i_t)}T_{N.i_1.....i_{t-1}}$, and so forth until ultimately the state described by $\tau_{(N.i_1.....i_t,N.i_1)}G_N$ is reached. We call this a *path from $T_{N.i_1.....i_t}$ to* $\tau_{(N.i_1.....i_t,N.i_1)}G_N$, or a path from $T_{N.i_1.....i_t}$ to G_N when we can avoid confusion. We refer to the triple ($N.i_1.....i_t$,$T_{N.i_1.....i_t}$, $\tau_{(N.i_1.....i_t,N.i_1)}G_N$) as an *input state triple*. We say that (M,T,μG) *precedes* (R,S,τG) if they are on the same path and R is a prefix of M.

To see how this works, consider the encrypt-decrypt protocol again. We start with goal word Z, with label 1. Consider solution 1.2, which used Protocol Rule 1, which says that if the intruder knows X and Y, he or she can produce e(X,Y), to deduce that if Z = e(X,Y) the intruder can learn Z if he or she knows X and Y. In this case $\sigma_{1.2}$ is the substitution Z/e(X,Y). Suppose that we apply Protocol Rule 1 to Y again, to obtain solution 1.2.2, in which the intruder can learn Y if Y = e(X_1,Y_1) and the intruder knows X_1 and Y_1. In this case $\sigma_{1.2.2}$ = Y/e(X_1,Y_1), and $\tau_{1.2.1}$, the composition of $\sigma_{1.2}$ and $\sigma_{1.2.1}$, is Z/e(X,e(X_1,Y_1)). The input state triple is (1.2.2, {X_1,Y_1,X},e(X,e(X_1,Y_1))).

4 How Languages are Represented and Verified in the Protocol Analyzer

A language rule is represented in the Protocol Analyzer database as a clause of the form

languagerule(N,langmember(W,*Langname*),
Conditions)

where N is an integer identifying the rule, W is a word, and *Conditions* is a set of conditions, which may include conditions saying that certain subterms of of W are members of languages. Thus, an example of a language rule would be

```
languagerule(5,langmember(e(A,B),seskey),
    langmember(B,seskey)).
```

which would be the Analyzer's internal representation of the language rule

Seskey → e(**L**,**Seskey**).

Since for the remainder of this paper, we will be describing the way in which the Protocol Analyzer deals with languages internally, we will use this internal representation of these language rules from now on.

The exact way in which language membership is verified is described in [5], so we do not go into detail here. Briefly, the outline is this.

A word belongs to a language *Lang* if and only if it is of the form σW, where

languagerule(N,langmember(W,*Lang*),C)

is a language rule, σW is irreducible, and σC is true. The condition C consists of the conjunction of conditions of the form langmember(W,*Lang*), not($W = V$), and lookedfor(Y), where Y is a subterm of W[2]. The first condition is self-explanatory. The second, not($W = V$), is interpreted to mean that there is no unifier σ of W and V so that σ is the identity on W (that is, V does not subsume W). The third, lookedfor(Y), is interpreted to mean that the intruder has not yet learned the word Y.

We use these facts to implement two Prolog procedures. One, expandconditions$_{\text{allsubs}}$, given a condition C returns a complete set **S** of substitutions σ and conditions E such that E implies σC. By complete we mean that, if τ is a substitution, then there is a (possibly empty) subset **T** of S such that, if $(\sigma_i,E_i) \in \mathbf{T}$, then $\tau = \mu_i\sigma_i$ and τC holds if and only if the logical disjunction of all $\mu_i E_i$ in **T** holds. The other procedure, expandconditions$_{\text{always}}$, given a condition C returns a condition E such that E implies C.

Expandconditions$_{\text{allsubs}}$ is computed as follows. For each occurrence of langmember(X,L) in a condition C where X is not a variable, it finds a rule languagerule(M,langmember(Y,L),D), and the most general unifier τ of X and Y. It then replaces langmember(τX,L) in τC with τD. It continues making these substitutions and replacements until no further nonvariable occurrences of langmember(X,L) can be found. The resulting condition is E, and this together with the substitution σ obtained by composing the most general unifiers obtained and restricting to the variables in C is called an *expansion pair* (σ,E). When queried repeatedly, expandconditions$_{\text{allsubs}}$ will produce the set of all expansion pairs. If Expandconditions$_{\text{allsubs}}$ finds no such unifiers, it produces the single expansion pair (ι,C).

As an example, we consider the language enckey described below, and consider the condition langmember(e(W,Z),enckey). The three language rules stored as Prolog clauses are of the form:

```
languagerule(1,
    langmember(e(X,key(A)),enckey),ok).
languagerule(2,langmember(e(X,Y),enckey),
    langmember(Y,enckey)).
languagerule(3,langmember(d(X,Y),enckey),
    langmember(Y,enckey)).
```

For the first application of expandconditions$_{\text{allsubs}}$, we unify e(W,Z) with e(X,key(A)) from Rule 1. The resulting condition is ok, that is, e(X,key(A)) is always in the language. For the second, we unify e(W,Z) with e(X,Y) from Rule 2 to obtain the condition langmember(W,enckey). In the case of Rule 3, we fail to unify d(X,Y) with e(X,Y). Thus, there are two expansion pairs produced: (Z/key(A),ok) and (ι,langmember(Z,enckey)) where ι is the identity substitution.

Expandconditions$_{\text{always}}$ is computed as follows. As in the case of expandconditions$_{\text{allsubs}}$, for each occurrence of langmember(X,L) in a condition C where X is not a variable, it finds a rule languagerule(M,langmember(Y,L),D), and a most general unifier τ of X and Y. However, it only succeeds if such a τ can be found that is the identity on C, that is, if Y *subsumes* X. If it does succeed, it replaces langmember(τX,L) = languagerule(X,L) in τC = C with τD. It continues making these substitutions and replacements until no further nonvariable occurrences of langmember(X,L) can be found. It computes all conditions that can be calculated this way, and returns the disjunction of these conditions.

Consider again the language enckey and the condition langmember(e(W,Z),enckey). For the first language rule, e(X,key(A)) does not subsume e(W,Z) because the substitution Z/key(A) is not the identity on Z. On the other hand, for the second language rule, the substitution is the identity. For the third language rule, there is no unifier. Thus the final result of expandconditions$_{\text{always}}$ is langmember(Z,enckey). This condition will imply langmember(e(W,Z),enckey) no matter what substitutions are made to W and Z.

We now use the procedures expandconditions$_{\text{allsubs}}$ and expandconditions$_{\text{always}}$ to produce a proof that knowledge of a member of a language implies previous knowledge. Our strategy is, for each language rule N defining a word W, to attempt to construct paths to

[2] A user defining a language actually has more leeway than this in defining conditions, but this describes the form of the conditions generated by the procedure described in this paper.

W, working backwards from W. We identify the state in which the intruder knows W as goal N, corresponding to our notation in Section 3. We use a breadth-first search strategy, first finding all states that can immediately precede goal N, then the states that can immediately precede each of those states, and so forth.

Each time we produce an input state, we attempt to prove that it contains a member of the language for all possible substitutions making W a member of the language. This is done as follows, by a procedure we call the *language membership verification procedure*. Let $(N.i_1.....i_k, T_{N.i_1.....i_k}, \sigma_{(N.i_1.....i_k,N.i_1)}\{W,C\})$ be an input state triple produced by a search for $\{W,C\}$. We would like to show that, whenever $\sigma_{(N.i_1.....i_k,N.i_1)}W$ is a member of *Lang*, then there is a word known by the intruder in $T_{N.i_1.....i_k}$ that is a member of *Lang*. We begin by determining all cases in which $\sigma_{(N.i_1.....i_k,N.i_1)}C$ can hold. We do this by invoking expandconditions$_{\text{allsubs}}$ on $\sigma_{N.i_1.....i_k}C$. For each expansion pair (τ, E) produced, we look at each word τV in $\tau T_{N.i_1.....i_k}$. We execute expandconditions$_{\text{always}}$ on langmember(τV,*Lang*) to produce a condition F that implies langmember(τV,*Lang*). We then attempt to show that E implies F. If, for each expansion pair (τ, E), there is some τV such that we can prove that this holds, then we will have proved our result for the input state $T_{N.i_1.....i_k}$. We mark the solution $S_{N.i_1.....i_k}$ as a success and do not attempt to prove the result for any solution preceding $S_{N.i_1.....i_k}$. Otherwise, we use the Analyzer to produce all solutions that can immediately precede $S_{N.i_1.....i_k}$ and perform the language membership verification procedure on the input state for each solution.

We continue in this fashion until we have either shown that all paths to W must contain a member of the language *Lang*, we encounter a path from an initial state to W that cannot be proved to contain a member of the language, or we encounter a path of length Q that cannot be proved to contain a member of the language, where Q is a parameter maintained by the system. In the first case, we will have succeeded in proving that intruder knowledge of W implies previous intruder knowledge of W, and in the other two cases we will have failed.

As an example, consider the language enckey again, and consider the second language rule, which has language member e(W,Z) with condition langmember(Z,enckey). Suppose that we ask the analyzer how the intruder can find e(W,Z), and it tells us that this can be done if the intruder knows d(R,Z). If we label e(W,Z) with the integer 1, the corresponding input triple will be (1.1, d(R,Z), {e(W,Z), langmember(Z,enckey)}). Computing expandconditions$_{\text{allsubs}}$ on langmember(Z,enckey), we will obtain the single expansion pair (ι,langmember(Z, enckey)). For this expansion pair, we attempt to compute expandconditions$_{\text{allsubs}}$ on langmember(d(R,Z),enckey). The only rule we can apply is languagerule(3, langmember(d(X,Y),enckey), langmember(Y,enckey)). This results in the condition langmember(Z,enckey). Since langmember(Z,enckey) implies langmember(Z,enckey), we are done.

A more detailed description of this process, with further examples and an outline of how it is implemented in Prolog, is given in [5].

5 How the Protocol Analyzer Generates Languages

5.1 Overview of this Section

We will present the Protocol Analyzer's language generation procedure in the following way. First, we will describe the general procedure for generating languages. Then we will describe in detail the various types of language rules that can be generated when we fail to show that a state contains a word belonging to the language we are trying to define. Once this is done, we will focus more broadly and describe the language generation process itself, dividing it into stages and describing each stage in detail.

5.2 How Rules are Generated

The strategy the Protocol Analyzer uses to generate languages is to start with one language rule, supplied by the user. It then attempts to prove the language unreachable by proving that knowledge of a word in the language implies previous knowledge of the word in that language. In each case in which it fails to do so, it either creates a rule that implies that one of the words the intruder must know previously is in the language, or modifies an old rule so that the particular word that is being verified is no longer in the language. The Analyzer now attempts to prove the new language unreachable, and adds or modifies rules as before. It continues this process until it either succeeds in proving the language unreachable or is unable to create any new rules. There is also the possibility that the Analyzer may get into an infinite loop, in which case it will fail after a certain number of iterations, which can be specified by the user.

The only input required by the user is to name the language, to input the first language rule, called the *seedword rule*, and to choose the search depth and strategy. The procedure for generating the seedword

rule is fairly straightforward. Languages usually arise out of the user's trying to prove a particular word unreachable. What the user often finds instead is a set of words that contain that word, or a portion of that word, as a subword, and which defines the language. We call the original word the user is trying to find the *seedword* of the language. Thus we begin by having the user specify the seedword S. In some cases, the seedword may contain a subword that is being looked for by the intruder. We allow the user to specify this one condition. Thus, if the user is trying to find out how to find e(X,Y), where Y is not known by the intruder, and specifies the name "encrypt" for the language, the Analyzer will construct the initial seedword rule

```
languagerule(1,langmember(e(X,Y),encrypt),
    lookedfor(Y)).
```

The basic scenario for generating rules is this. Suppose that we are given a rule of the form

languagerule(N,langmember(W,$Lang$),C)

and we are attempting to prove that, if C holds, then every path leading to a state **S** in which the intruder knows W and the conditions in C hold contains a state in which the intruder knows a member of $Lang$. That is, for each path, we want to show that there is an input triple (M,T, $\sigma\{W,C\}$) such that T contains a word of $Lang$. We attempt to prove that T contains a member of $Lang$ by running expandconditions$_{\text{allsubs}}$ on σC, and for each expansion pair (E,τ) generated, attempt to prove that E implies that τT contains a word in $Lang$ by showing that, for at least one X in τT, E implies the result of running expandconditions$_{\text{allsubs}}$ on langmember(X,$Lang$). If we fail to do so, we generate a new rule.

The way in which the new rule is generated depends upon the structure of the words in τT. We classify rules as of type I, II, or III, depending upon how they are generated.

Briefly, a rule of type I is generated when an input word is generated containing a word Z known to be a member of the language as a subword. We replace Z or some subword containing Z with a variable Y, and generate a rule or set of rules saying that this word is in the language as long as Y is in the language.

A rule of type II is generated when we find that σW can be obtained for some σ. Let R be a subword occurring in σW such that R is in the language. Then $R = \mu U$, from some language rule

languagerule(N,langmember(U,$Lang$),C').

We add to the condition C' the condition not($U = R$) (that is, we are saying that U in general cannot be found, except possibly for the case $U = R$).

A rule of type III is generated when W contains a lookedfor word Y, and the input word contains Y but not W. We generate a new rule or set of rules that say that the input word is in the language as long Y is a lookedfor word.

These rules are described in detail below.

5.3 The Different Types of Rules

5.3.1 Rules of Type I

We have already encountered rules of type I in Section 2. Suppose that τT contains a word U either containing W as a subword or a word X such that langmember(X,$Lang$) appears in E. We can make τT contain a member of $Lang$ by adding the rules

languagerule(N_i, langmember(V_i,$Lang$),
    langmember(Y_i,$Lang$))

where the V_i and Y_i are created as follows.

Let V be the smallest subterm of U containing W (or X) such that membership of V in $Lang$ would imply membership of U in $Lang$. If there is more than one such subterm, choose the one with the least occurrence. Let $i_1.i_2...i_k$ be the least occurrence of W (or X) in V. Let V_1 be the result of replacing the term occurring at i_1 in V by the variable Y_1. The language rule

languagerule(N_1, langmember(V_1,$Lang$),
    langmember(Y_1,$Lang$))

will guarantee that V is in $Lang$ as long as the term occurring at i_1 is. Similarly, if Z is the term occurring at $i_1.i_2...i_j$ in V, where $j < k$, let V_{j+1} be the result of replacing in Z the term occurring at $i_1.i_2...i_j.i_{j+1}$ with the variable Y_{j+1}. The language rule

languagerule(N_{j+1}, langmember(V_{j+1},$Lang$),
    langmember(Y_{j+1},$Lang$))

will guarantee that Z is in $Lang$ as long as the term occurring at $i_1.i_2...i_j.i_{j+1}$. Together, all these rules will imply that V is in $Lang$ as long as W (or X is), and hence that U is in $Lang$ as long as W or X is.

We call rules generated in this way *rules of type I*.

For example, suppose that we started out with the seedword rule

```
languagerule(1,langmember(e(X,Y),encrypt),
    ok)
```

and the Analyzer discovered an input state triple (M,T,{e(X,Y),ok}) where T is the state in which the intruder knows e(R,(Q,d(Z,e(X,Y)))), where (_,_) is the concatenation function. The result of applying expandconditions$_{\text{allsubs}}$ to the condition lookedfor(Y) is the expansion pair (ι,ok), where ι is the identity. Clearly, ok does not imply that e(R,(Q,d(Z,e(X,Y)))) is a member of encrypt, so we attempt to generate a rule of Type I. The smallest subterm of e(R,(Q,d(Z,e(X,Y)))) whose membership in encrypt would imply membership of the whole word in encrypt is (Q,d(Z,e(X,Y))). So in this case we would generate two rules:

```
languagerule(2,langmember((Q,Y1),encrypt),
    langmember(Y1,encrypt)).

languagerule(3,langmember(d(Z,Y2),encrypt),
    langmember(Y2,encrypt)).
```

Notice that it would also be possible to generate the single language rule

```
languagerule(2,
    langmember((Q,d(Z,Y1)),encrypt),
        langmember(Y1,encrypt)).
```

However, we prefer to generate the multiple rules, first, because they result in a larger language, and secondly, because they result in simpler, more uniform-appearing languages for which it is easier to develop faster algorithms for verifying membership.

5.3.2 Rules of Type II

In a number of cases, it will not be possible to produce a rule or rules of Type I. But, it may be that $\tau\sigma W$ contains a subterm V satisfying

languagerule(Q,langmember(X,$Lang$),C')

for some language rule, that is, there is a substitution μ such that V = μX and $\mu C'$ holds. If μ is not the identity, we can modify the language rule to

languagerule(Q, langmember(X,$Lang$),
(C', not(X=V))).

We also have the option, if C' contains a condition of the form lookedfor(Z), and Z, and $U = \mu Z$, of modifying the language rule to be

languagerule(Q, langmember(X,$Lang$),
(C',not(Z=U)).

The latter can be helpful if language rules of type III are to be generated, as we will see in the next section.

We call either type of rule a *rule of Type II.*

For example, suppose we are trying to generate the language encrypt2 from the seedword rule

```
languagerule(1,langmember(e(X,Y),encrypt2),
    lookedfor(Y))
```

and that the Analyzer generated the input triple (M,ϕ,{e(key(A),rand(A,N),ok}) where ϕ is the empty set. Then, depending upon which strategy we are using, we can generate one of the following rules of type II that will guarantee that e(key(A),rand(A,N)) no longer satisfies Rule 1:

```
languagerule(1,langmember(e(X,Y),encrypt2),
    (lookedfor(Y),
        not(e(X,Y) = e(key(A),rand(A,N)))))
```

or

```
languagerule(1,langmember(e(X,Y),encrypt2),
    (lookedfor(Y),
        not(Y = rand(A,N)))).
```

At this point in the implementation of the Protocol Analyzer, we restrict ourselves to modifying seedword rules and rules of Type III when we generate rules of Type II. Rules of Type III are described in the next section.

5.3.3 Rules of Type III

A third type of rule, which arises more rarely than the other two, is used in only in the case in which C contains a condition lookedfor(Y) where Y is a subterm of W. Suppose that we have an input triple (M,T,σ{W,C}) and an expansion pair (τ,E) such that τT contains a word U containing $\tau\sigma Y$ as a subterm, but U does *not* contain $\tau\sigma W$. This can be used to generate what we call *rules of Type III.*

Our procedure for generating rules of Type III is similar to that for generating rules of Type I.

We add the rules

languagerule(N_i, langmember(V_i,$Lang$),
langmember(Y_i,$Lang$))

where the V_i and Y_i are created as follows.

Let V be the smallest subterm of U containing $\tau\sigma Y$ such that membership of V in $Lang$ would imply membership of U in $Lang$. If there is more than one such subterm, choose the one with the least occurrence. Let $i_1.i_2...i_k$ be the least occurrence of V in U. Let V_1 be V after the term occurring at i_1 has been replaced by

the variable Y_{i_1}. Similarly, if Z is the term occurring at $i_1.i_2...i_j$, where $j < k$, let V_{j+1} be Z after the term occurring at $i_1.i_2...i_j.i_{j+1}$ has been replaced with the variable Y_{j+1}.

For j from 1 to k-1, we add the rules

languagerule(N_j, langmember(V_j,$Lang$),
langmember(Y_j,$Lang$)).

For $j = k$, we add the rule

languagerule(N_k, langmember(V_k,$Lang$),
(lookedfor(Y_k), C))

where $Y_k = Y$ and V_k is the smallest subterm of U not equal to Y containing Y.

The remaining part of the condition C is constructed as follows. Suppose that the current form of the seedword rule is

languagerule(1, langmember(W,$Lang$),
lookedfor(X)).

Then C is empty. On the other hand, if the current form of the seedword rule is

languagerule(1, langmember(W,$Lang$),
lookedfor(X),D),

where D is the concatenation of conditions of the form not($X = S$), then C is set equal to D. If the seedword rule is of any other form, the procedure fails.

For example, suppose that, we attempted to define a language with the seedword rule

```
languagerule(1,langmember(e(X,Y),encrypt2),
    lookedfor(Y))
```

and at some point this had been replaced with the rule of Type II

```
languagerule(1,langmember(e(X,Y),encrypt2),
    (lookedfor(Y),not(Y = rand(A,N)))).
```

Suppose that the Analyzer found an input state triple of the form (N, T, {e(X,Y), lookedfor(Y), not(Y = rand(A,N))}). Suppose, furthermore, that T was found to contain a word d(Z,Y) for some Z. Then we could construct a rule of Type III of the form

```
languagerule(3,langmember(d(X,Y),encrypt),
   (lookedfor(Y),not(Y = rand(A,N)))).
```

5.4 Strategies for Generating Rules

Our next problem is to choose a strategy for generating rules. In many cases we will have a choice between generating a rule of type I, II, or III. The strategy we have chosen is to prefer rules of type I over rules of type II, since adding rules of type I makes the language larger (which is preferable), and adding rules of type II makes it smaller. However, although rules of type III also extend the language, it turns out that they in turn must satisfy additional constraints in order to make them consistent with the initial seedword rule, which in turn also limits the size of the language. Thus we generate rules of type III only as a last resort.

The use of rules of Type III puts a constraint on the generation of rules of Type II in the following way. Suppose that we have generated a rule of type II

languagerule(N, langmember(W,$Lang$),
$Conditions$)

where $Conditions$ contains a condition of the form not($W = Z$). Suppose that next we generate a rule of type III

languagerule(M,langmember(V,$Lang$),
$Conditions$).

If W contains variables not in V, the rule of type III generated may be vacuous. For example, suppose that the rule of type II is

```
languagerule(1,langmember(e(X,Y),encrypt),
    (lookedfor(Y), not(e(X,Y) = e(key(A),Y)))
```

and the rule of type III is

```
languagerule(3,langmember(d(Z,Y),encrypt),
    (lookedfor(Y),
        not(e(X,Y) = e(key(A),Y)))).
```

The second language rule says that a word in the language must satisfy the condition that there is no X such that e(X,Y) = e(key(A),Y), which is patently false. The fact that the first language rule says that this is the case for a particular value of X does not imply the result for the second language rule.

We get around this by using a different strategy for computing rules of Type II when we expect to encounter rules of Type III. Given a rule

languagerule(N,langmember(W,$Lang$),
$Conditions$)

where *Conditions* contains a condition of the form lookedfor(Y), when we find that $\tau\sigma W$ can be found, where $\tau\sigma W$ is not the identity, we augment *Conditions* by the condition not($Y = \tau\sigma Y$). Now, when a rule of Type III

languagerule(Q,langmember(V,*Lang*),
Newconditions)

is generated, it will contain Y as a subterm, so it will be possible to use the first rule to show that the second holds. We call the original strategy for generating rules of type II Strategy 1, and the new strategy Strategy 2.

It is advisable to use Strategy 1 when possible, since it generally leads to bigger languages. On the other hand, there are a number of cases in which languages cannot be generated without use of Strategy 2. There are also a few cases in which Strategy 2 might be preferable since it puts the conditions directly on the looked-for word, which is the word the user was originally trying to determine if the intruder could find. Since we cannot determine in advance which strategy would be preferable, we give the user the choice. If the user states no preference, the Analyzer tries Strategy 1. If that fails, the Analyzer uses Strategy 2.

5.5 The Procedure for Generating a Language

Languages are generated by iterating a three step-process. These are: input word generation, verification, and rule generation. We describe each of these in detail below.

5.5.1 Input Word Generation

In the input word generation step, we take each rule

languagerule(N, langmember(W,*Lang*),
Conditions)

to which the input word generation step has not previously been applied, and use the Protocol Analyzer to find all conditions under which the intruder could learn W, discarding any results that violate the conditions in *Conditions*. For each solution that contains nonempty local state variables as input, the Analyzer is used to find the conditions in which a state in which those variables have these values could be reached. This process is iterated until either the only states remaining to be queried have no nonempty state variables as input, or some limit on state space depth defined by the user has been reached. Solutions are identified using the decimal notation described in Section 3, as $N.N_1.N_2.$ ... N_k, where N is the integer identifying the rule.

Our reason for having the Analyzer query only nonempty local state variables, and not words known by the intruder, is to keep the state space relatively small. If we allowed the Analyzer to query words as well, we would risk running into the very state space explosion that we are trying to control. We have used the technique of restricting our queries to nonempty state variables with success on a number of different protocols, and this seems to be an optimal solution, at least in the early stages in the analysis. In the later stages, in which state space explosion is already well under control, it may be preferable to use a less restrictive policy.

5.5.2 Verification

In this step, the output generated during the Input Word Generation step is examined to show that, for every case in which a word W belongs to the language *Lang* according to a rule

languagerule(N,langmember(W,*Lang*),C)

every path to W requires intruder knowledge of a member of *Lang*. Paths for which we fail to produce such a result are saved for the rule generation section. We construct a set $FAIL_N$ of the input states appearing in such paths, together with the conditions for which we failed to prove language membership, as follows.

We start with two sets, $NODES_N$ and $FAIL_N$. Initially, $FAIL_N$ is empty, while $NODES_N$ is the set of all input state triples (M, T, $\sigma\{W,C\}$) produced in the input word generation step. For each element (M, T, $\sigma\{W,C\}$) of $NODES_N$, we attempt to use the verification procedure outlined in Section 4 to prove that, in each case where σW is a member of *Lang*, then T must contain a member of *Lang*. In other words, we use the expandconditions$_{\text{allsubs}}$ procedure to produce the set of expansion pairs (E,τ) of the condition C, and for each such pair, we attempt to prove that there is a word V in τT such that E implies that V is in *Lang*. If we cannot prove the result, we delete the triple from $NODES_N$, but we add to $FAIL_N$ all 4-tuples (M,τT,$\tau\sigma W$,E) for which our attempt to prove membership of an element of τT failed. If we can prove the result, we delete (M, T, $\sigma\{W,C\}$) from $NODES$, as well as all triples (P,S,$\mu\{W,C\}$) where P precedes M. We also delete all such (P,S,τW,E) from $FAIL_N$. We continue until the set $NODES_N$ is empty.

Lemma: If, for each rule N, $FAIL_N$ is empty once the procedure described above completes, then we have successfully proved the language unreachable.

Proof: If we can prove that, if $FAIL_N$ is empty when the procedure completes, then each path to N

contains at least one state in which the intruder must know a member of the language, then we are done.

Suppose that we are given an input state triple (P,S,$\mu\{W,C\}$). Then, we have either shown that, for all expansion pairs (E,τ) of C, there is a member of τS that can be shown to be a member of $Lang$, or there is a triple (M,P,$\sigma\{W,C\}$) for which this is true and for which P precedes M. In the first case, we have shown that, for all possible substitutions τ making τW a member of $Lang$, then τS contains a member of $Lang$. In the latter, we have shown that there is a M preceded by P for which this is true. Since the path from P to N must pass through M, in either case we have shown that any path containing P can be shown always to contain a member of $Lang$. □

If $FAIL_N$ is empty for all N, the algorithm proceeds to the Optimization Step. If there is at least one nonempty $FAIL_N$, we proceed to the Rule Generation Step.

5.5.3 Rule Generation

In this step, we examine the elements of each nonempty $FAIL_N$ to see if we can generate rules that will guarantee unreachability of the language L.

We search $FAIL_N$ in a top-down fashion, so that, if Q precedes M, then (M,T,σW,E) is examined first. Let languagerule(N, langmember(W,$Lang$), C) be a language rule and (M, T, σW, E) be a 4-tuple on a path to W from $FAIL_N$. We determine if T contains a word V containing σW as a subterm, or if E contains a condition langmember(X,$Lang$) such that V contains σX as a subterm. If either of those is the case, we construct the appropriate rule of Type I and enter it into the database. We also delete from $FAIL_N$ all 4-tuples (Q, S, τW, G) such that Q precedes M. If we cannot construct a rule of Type I, and M is not an initial state, we keep the tuple in $FAIL_N$.

If M is an initial state, that is, there is no state S preceding M, we attempt to add a rule of Type II as follows. We examine σW and attempt to see if σW contains a subword V, such that, if

languagerule(Q,langmember(X,L),
$Conds$)

is the seedword rule or a rule of Type III, then $V = \mu X$ for some μ, and $\mu Conds$ is implied by E. If it does, we create a rule of Type II according to the procedure described in Section 5.3.2. We delete (M,T, σW,E) from $FAIL_N$.

If M is an initial state, and we were not able to generate a rule of Type I or of Type II from it, and we are using Rule Generation Strategy 1, then the 4-tuple remains in $FAIL_N$, and the language generation procedure fails, since, because there are no states preceding M, it will be impossible to remove (M,T, σW,E) from $FAIL_N$. If, however, we are using Rule Generation Strategy 2 and E contains a condition of the form lookedfor(Z) for some Z, we attempt to generate a rule of Type III according to the procedure described in Section 5.3.3. If we succeed we delete (M,T, σ,E) from $FAIL_N$.

We continue in this fashion until every node in $FAIL_N$ has been examined or removed. We now remove the remaining nodes from $FAIL_N$ in the following fashion. If, for a given M, all tuples of the form (M,T,σW,E) have been deleted, we also delete all tuples (Q,S,μW,F) such that Q precedes M. If at the end, $FAIL_N$ is empty for all N, we proceed to the input word generation step, to generate input for the rules we have created and modified. If $FAIL_N$ is nonempty for any N, then we have failed to generate rules to cover all cases, and the procedure terminates, reporting failure.

5.5.4 Optimization

The optimization step takes place only after the verification step completes successfully, having verified language membership for all paths. In the optimization step, we take each rule

languagerule(N,langmember(X,L),
$Conds$)

one by one, and delete it from the rule database. We then attempt to use the remaining rules to show that, if X satisfies $Conds$, then X is a member of L. If this is the case, then the rule is removed permanently, since it has been proved to be redundant. It this is not the case, the rule is returned to the database. We apply this procedure to each rule in turn.

5.6 An Example

In this section we present an example to illustrate how the language generation procedure works.

We begin with the following augmented version of the Encrypt-Decrypt protocol.

Protocol Rule 1
If the intruder knows X and Y, then he or she can find e(X,Y), where e(X,Y) denotes the encryption of Y with key X.

Protocol Rule 2
If the intruder knows X and Y, then he or she can find d(X,Y), where d(X,Y) denotes the decryption of Y with key X.

Protocol Rule 3
If the intruder sends the name node(A) to a random number server, the server will respond with e(key(host(A)),rand(server,N)), where rand(server,N) is a random number generated by the server.
Protocol Rule 4
If the intruder knows d(X,Y), he or she can produce e(X,Y).
Protocol Rule 5
The intruder knows all names node(A) initially.

Suppose that a user wishes to find out under what circumstances the word e(W,Z) can be learned, where Z is a word not known by the intruder. We will illustrate the use of strategy 2; strategy 1 would not succeed in this case. The Analyzer begins by defining the seedword language rule

```
languagerule(1,langmember(e(W,Z),encrypt),
    lookedfor(Z)).
```

It next finds all possible input triples to e(W,Z) that do not violate the conditions lookedfor(Z). From Protocol Rule 1, we have

(1.1,{X_1,d(X_1,e(W,Z))},{e(W,Z),lookedfor(Z)}).

From Protocol Rule 2, we have

(1.2,{X_1,e(X_1,e(W,Z))},{e(W,Z),lookedfor(Z)}).

From Protocol Rule 3, we have

(1.3,{node(A)},{e(node(A),rand(server,N)),
lookedfor(rand(server(N))}}).

From Protocol Rule 4, we have

(1.4,{d(X,W)},{e(W,Z),lookedfor(Z)}).

Protocol Rule 5 produces no input triples.

We do not query any of these input triples any further to produce new triples, since none of them contains local state variables.

We now attempt to determine which input triples contain words belonging to the language encrypt by Language Rule 1. We begin by performing expandconditions$_{\text{always}}$ on σlookedfor(Z) for the substitution σ in each triple. For 1.1, 1,2, and 1.4, $\sigma = \iota$ and there is only one expansion pair, (ι,lookedfor(Z)). For 1.3, the one expansion pair is (ι,lookedfor(rand(server,N)).

For each result of applying expandconditions$_{\text{always}}$, we attempt to use expandconditions$_{\text{allsubs}}$ to prove that knowledge of a word from encrypt implies previous knowledge of a word from encrypt. It is clear that there is only one input triple containing a word that is subsumed by e(W_1,Z_1) from language rule 1. This is 1.2, for which the language member is e(X_1,e(W,Z)). Applying expandconditions$_{\text{allsubs}}$ to langmember(e(X_1,e(W,Z)),encrypt) yields the condition lookedfor(e(W,Z)). Since lookedfor(Z) does not imply lookedfor(e(W,Z)), the four-tuple (1.2,{X_1,e(X_1,e(W,Z))},{e(W,Z),lookedfor(Z)}, lookedfor(e(W,Z))) goes into $FAIL_1$. Thus $FAIL_1$ consists of:

(1.1,{X_1,d(X_1,e(W,Z))},e(W,Z),lookedfor(Z))
(1.2,{X_1,e(X_1,e(W,Z))},e(W,Z),lookedfor(Z))
(1.3,{node(A)},e(node(A),rand(server,N)),
lookedfor(rand(server(N))))
(1.4,{d(X,W)},e(W,Z),lookedfor(Z)).

We look at the first member of $FAIL_1$. It contains the seedword as a subword, so we can use it to generate a rule of Type I. The smallest subterm of d(X_1,e(W,Z)) whose membership in encrypt would imply membership of the whole term in encrypt is the word itself, so the rule becomes

```
languagerule(2,langmember(d(W,Z),encrypt),
    langmember(Z,encrypt)).
```

The next member of $FAIL_1$ also contains the seedword as a subword, The smallest subterm of e(X_1,e(W,Z)) whose membership in encrypt would imply membership of the whole term in encrypt is again the word itself, so the rule becomes

```
languagerule(3,langmember(e(W,Z),encrypt),
    langmember(Z,encrypt)).
```

Looking at the next member, 1.3, we see that the only input word is node(A). This does not contain the seedword or any lookedfor word, so we create a rule of Type II by modifying language rule 1 as follows:

```
languagerule(1,langmember(e(W,Z),encrypt),
    (lookedfor(Z),not(Z = rand(server,N)))).
```

Looking at the last member, 1.4, we see that, although no input word contains a seedword, the input word d(X,W) contains the lookedfor word W. Thus the Analyzer creates a rule of Type III. Since the seedword rule now contains an exception, the new Type III rule must contain the same exception.

```
languagerule(4,langmember(d(W,Z),encrypt),
    (lookedfor(Z),not(Z = rand(server,N)))).
```

Once the Analyzer has finished generating these new rules, it now repeats the procedure with the new rules. We leave it as an exercise to the reader to determine

that in the next round no new rules will be generated and the language is complete.

The alert reader may notice that this language is a little smaller than necessary. The language does not contain any words of the form d(X,rand(server,N)), where rand(server,N) is a lookedfor word. If we define a new language with seedword d(X,Y), where Y is a lookedfor word, the reader can verify that this language will contain all d(X,Y) where Y is a lookedfor word. Moreover, if this language had been generated and verified first, the rule of type III for the language encrypt would never have been generated.

5.7 Variations on the Procedure

For the sake of the exposition, we have described a procedure in which a given step does not begin until the previous step has completed. However, for the sake of efficiency, it is possible to run some of the steps concurrently. This is what we are doing in the current implementation of the procedure. When the Analyzer fails to prove membership for any words in a path to W during the verification step, it immediately generates a rule which is added to the database. If the rule is a modification of an old rule (that is, a rule of Type II), the old rule and the set of paths to it constructed in the input word generation step is deleted. This allows us to use the newly created rules to verify language membership for the remaining paths, and saves us the possible waste of generating redundant or duplicate rules.

In our current implementation, we also attempt to generate a rule of Type I immediately when we fail to prove that an input state pair (T,σ) always contains a member of the language, instead of first examining each descendant of (T,σ). This saves some time, at the potential cost of introducing unnecessary language rules. However, the results we have obtained so far have indicated that this is a reasonable tradeoff.

Another improvement can be realized in the way that conditions for the input triples are calculated. For the purposes of this paper the conditions are calculated by applying substitutions to the original goal condition. However, protocol rules may also have conditions on their input words. When a protocol rule is used to create an input triple, these conditions are inherited by the input words. Thus, instead of creating a triple (M,T,$\sigma\{W,C\}$) we can create (M,T,{ σW, σC,C'}) where C' is the set of conditions on T imposed by the protocol rule.

6 Experiences Using the Language Generator

We have run the language generator on several protocols: in particular, on the Needham-Schroeder public-key protocol [6], for which we were able to reproduce the spoofing attack found by Gavin Lowe [2, 3], and on the Woo and Lam secure boot protocol [8]. An account of our analysis of the Needham-Schroeder protocol may be found in [4]. We have found it to be quite helpful, not only in speeding up the language search, but in avoiding confusion, especially in the generation of rules of types II and III. This was especially the case for the Woo and Lam secure boot protocol, which has a very complex message structure, not only using public keys to encrypt messages containing data encrypted by shared keys, but shared keys to encrypt messages containing data encrypted (or signed) by public (or private) keys. Our previous attempts to define languages by hand for this protocol had been very time-consuming and met with limited success. By using the language generator, although we still had to give some thought to the order in which languages were generated, we were able to find appropriate languages with a minimum of work.

7 Conclusion

In this paper we set out a procedure for automating language generation in the NRL Protocol Analyzer. Previously languages were defined by hand and then proved unreachable automatically. Now, both generation and proof of unreachability are done automatically together, saving both time and labor on the part of the user of the Analyzer.

Another advantage of using the automated language generator is that it produces languages with a very simple standard format. A word is a member of a language if it is equal to a certain term one of whose arguments is a member of the language, or it is equal to a certain term one of whose arguments is not known by the intruder, and the term (or the argument) is not equal to certain values. This simple format allows us to concentrate on building fast procedures for proving that a word is or is not a member of a language. Since proving language membership is at present one of the most time-consuming portions of the Analyzer, this should be of assistance to us in improving the Analyzer's performance.

An obvious question that comes to mind is: how helpful would the techniques that we have described in this paper be for proving unreachability results for

other formal systems making use of rewrite rules, including extensions of the NRL Protocol Analyzer? In our case, we found ourselves aided by the fact that the rewrite rules we used followed a very simple format: namely, they are all of the form

$G \rightarrow X$

where X is a variable appearing in G. Although the NRL Protocol Analyzer does not require all rewrite rules to be of this form, rules of this sort reflect the way in which most cryptographic algorithms operate. But this fact means that, if we ask the Protocol Analyzer how to find a word X, we will get a number of responses requiring the intruder's previous knowledge of a word containing X, thus explaining the preponderance of rules of Type I. It would be interesting to see if there were other classes of rules arising out of other types of applications that would give rise to different types of language generation strategies.

References

[1] D. Dolev and A. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[2] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[3] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *Proceedings of TACAS*. Springer Verlag, 1996.

[4] C. Meadows. Analyzing the Needham-Schroeder public-key protocol: A comparison of two approaches. submitted for publication, March 1996.

[5] C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, February 1996.

[6] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[7] P. Réty, C. Kirchner, H. Kirchner, and P. Lescanne. NARROWER: A new logic for unification and its application to logic. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, 1985*, pages 141–157. Springer Verlag LNCS 202, May 1985.

[8] T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. *IEEE Computer*, 25(1):39–52, January 1992.